A homage to Phil Wadler's "The essence of functional programming", POPL 1992

The *quick* essence of functional programming

Ralf Lämmel
Software Language Engineer
University of Koblenz-Landau
Germany

See here for source code and additional lectures:
http://sourceforge.net/apps/mediawiki/developers/index.php?title=Ralfs-channel9-lectures

Computing 42

```
(\x -> x+x) (10+11)
```

```
term42
 = App (Lam "x" (Add (Var "x") (Var "x")))
       (Add (Con 10) (Con 11))
```

```
> interp term42 []
42
```

Another interpreter for the lambda calculus -- ***to be extended by computational facets***.

Syntactic and semantic domains

```
type Name = String

data Term =
    Var Name
  | Con Int
  | Add Term Term
  | Lam Name Term
  | App Term Term

data Value =
    Wrong
  | Num Int
  | Fun (Value -> Value)

type Environment = [(Name, Value)]
```

`interp :: Term -> Environment -> Value`

Watch this!

The interpreter

```
interp :: Term -> Environment -> Value
interp (Var x) e = lookup x e
interp (Con i) e = Num i
interp (Add u v) e = add (interp u e) (interp v e)
interp (Lam x v) e = Fun (\a -> interp v ((x,a):e))
interp (App t u) e = apply (interp t e) (interp u e)
```

```
lookup :: Name -> Environment -> Value
lookup _ [] = Wrong
lookup x ((y,b):e) = if x==y then b else lookup x e
```

```
add :: Value -> Value -> Value
add (Num i) (Num j) = Num (i+j)
add _ _ = Wrong
```

```
apply :: Value -> Value -> Value
apply (Fun k) a = k a
apply _ _ = Wrong
```

Exercise: revise this interpreter to model environments as functions instead of lists of pairs.

Suppose we want to enable ...

- output in addition to the value result

```
Term -> Env -> (Value, Output)
```

- error messages instead of wrong values

```
Term -> Env -> Either String Value
```

- state transformation in addition to the value result

```
Term -> Env -> State -> (Value, State)
```

The impact on the signature of interpretation is highlighted.

Interpretation with state transformation: impact on interpretation function

```
interp (Var x) e s = (lookup x e, s)

interp (Con i) e s = (Num i, s)

interp (Add u v) e s0
  = let (v1,s1) = interp u e s0
        (v2,s2) = interp v e s1
    in add v1 v2 s2

interp (Lam x v) e s
  = (Fun (\a -> interp v ((x,a):e)), s)

interp (App t u) e s0
  = let (v1,s1) = interp t e s0
        (v2,s2) = interp u e s1
    in apply v1 v2 s2
```

Exercise: do the patch for output.

Step 1 of conversion to monadic style: parametrize in a type constructor for computations

```
interp :: Term -> Environment -> M Value
```

```
data Value =
    Wrong
  | Num Int
  | Fun (Value -> M Value)
```

```
lookup :: Name -> Environment -> M Value
add :: Value -> Value -> M Value
apply :: Value -> Value -> M Value
```

Specific monad-type constructors

- No effects.

```
type M a = a
```

- Produce output.

```
type M a = (a, String)
```

- Transform state.

```
type M a = State -> (a, State)
```

- ...

Terminology:
- a ... values
- *M* a ... computations

Eventually, we will use ***abstract*** datatype constructors.

Step 2 of conversion to monadic style: compose computations in chains

```
interp (Add u v) e =  add
                          (interp u e)
                          (interp v e)
```

Regular style uses functional decomposition.

```
interp (Add u v) e =  let a = interp u e in
                      let b = interp v e in
                      add a b
```

Sequential style in preparation of monadic style.

```
interp (Add u v) e =  interp u e >>= (\a ->
                      interp v e >>= (\b ->
                      add a b))
```

Compose computations in chains with “bind”.

```
interp (Add u v) e = do  a <- interp u e
                         b <- interp v e
                         add a b
```

Convenient **do** notation.

Ingredients of a ***monad***

```
type M
```

Construct computation from value type

```
return :: a -> M a
```

Get into the monad: construct computations from values

```
(>>=) :: M a -> (a -> M b) -> M b
```

Apply function to computation

The identity monad

```
type M a    = a
return a    = a
a >>= k     = k a
```

```
> interp term42 []
42
```

That's the same interpreter as the non-monadic baseline modulo partial evaluation.

CBV monadic style interpreter

```
interp :: Term -> Environment -> M Value
interp (Var x) e = lookup x e
interp (Con i) e = return (Num i)
interp (Add u v) e = interp u e >>= \a ->
                     interp v e >>= \b ->
                     add a b
interp (Lam x v) e
 = return (Fun (\a -> interp v ((x,a):e)))
interp (App t u) e = interp t e >>= \f ->
                     interp u e >>= \a ->
                     apply f a
```

Auxiliary functions

```
lookup :: Name -> Environment -> M Value
lookup _ [] = return Wrong
lookup x ((y,b):e)
 = if x==y then return b else lookup x e
```

```
add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num (i+j))
add a b = return Wrong
```

```
apply :: Value -> Value -> M Value
apply (Fun k) a = k a
apply f a = return Wrong
```

A riddle: There is no (>>=) here! Why is that?

Interpreter revision: return error messages instead of `Wrong`

```
termE = App (Con 1) (Con 2)
```

Baseline

```
> interp termE []

<wrong>
```

Revision

```
> interp termE []

<error: should be function: 1>
```

The error monad

```
data M a = Suc a | Err String
```

... or use `Either`

```
return a = Suc a
```

```
(Suc a) >>= k = k a
(Err s) >>= k = Err s
```

```
fail :: String -> M a
fail s = Err s
```

Special operation of this monad to signal errors.

Selective code replacement

Baseline

```
apply (Fun k) a = k a
apply f a = return Wrong
```

Revision

```
apply (Fun k) a = k a
apply f a = fail (
  "should be function: " ++ show f)
```

The role of show

```
instance Show Value where
  show Wrong = "<wrong>"
  show (Num i) = show i
  show (Fun _) = "<function>"

data M a = Suc a | Err String

instance Show a => Show (M a) where
  show (Suc a) = show a
  show (Err s) = "<error: " ++ s ++ ">"
```

Get out of the monad

Interpreter revision: read reduction count

```
> let test t = show (interp t [])

> putStrLn (test (Add (Con 21) (Con 21)))

Value: 42; Count: 1

> let z = Con 0

> putStrLn (test (Add (Add z z) Count))

Value: 1; Count: 2
```

The state monad

```
type M a = State -> (a, State)
type State = Int -- Reduction count
```

```
return a = \s0 -> (a, s0)
```

```
m >>= k
 = \s0 -> let (a,s1) = m s0 in k a s1
```

```
tick :: M ()
tick = \s -> ((), s+1)
```

```
fetch :: M State
fetch = \s -> (s, s)
```

Special operations of this monad to read and write state.

Getting out of the state monad

```
instance Show (M Value) where
  show m = let (a,s1) = m 0
           in "Value: " ++ show a ++ "; " ++
              "Count: " ++ show s1
```

Data extension and selective code replacement

```
data Term = ... | Count
```

```
interp Count e
 = fetch >>= \i -> return (Num i)
```

```
add (Num i) (Num j)
 = tick >>= \() -> return (Num (i+j))
```

```
apply (Fun k) a
 = tick >>= \() -> k a
```

(Catch-all cases omitted.)

Interpreter revision: produce output in addition to returning a value

```
termO
 = Add (Out (Con 41)) (Out (Con 1))
```

```
> let test t = show (interP t [])
> putStrLn (test termO)
(42,"41; 1; ")
```

The writer monad

```
type M a = (a, String)
```

```
return a = (a, "")
```

```
m >>= k = let (a,r) = m
              (b,s) = k a
          in  (b, r++s)
```

```
tell :: Value -> M ()
tell a = ((), show a ++ "; ")
```

Special operation of this monad to extend output.

Data extension (no selective code replacement)

```
data Term = ... | Out Term
```

```
interp (Out u) e
 =  interp u e >>= \a  ->
    tell a      >>= \() ->
    return a
```

Interpreter revision: evaluate nondeterministically

```
termL
 = App (Lam "x" (Add (Var "x") (Var "x")))
       (Amb (Con 2) (Con 1))
```

```
> let test t = show (interP t [])
> putStrLn (test termL)
[4,2]
```

The list monad

```
type M a = [a]
```

```
return a = [a]
```

```
m >>= k = [ b | a <- m, b <- k a ]
```

```
mzero :: M a
mzero = []
```

```
mplus :: M a -> M a -> M a
l `mplus` m = l ++ m
```

Data extension (no selective code replacement)

```
data Term
 = ... | Fail | Amb Term Term
```

```
interp Fail e
 = mzero
interp (Amb u v) e
 = interp u e `mplus` interp v e
```

The type class ***Monad***

Type parameter is of kind `*->*`

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  ...
```

Laws

```
return a >>= f   ≡ f a
m >>= return     ≡ m
(m >>= f) >>= g  ≡ m >>= (\x -> f x >>= g)
```

From implicit to explicit parameterization

```
interp :: Term
       -> Environment
       -> M Value
```

```
interp :: Monad m
       => Term
       -> Environment
       -> m Value
```

```
interp :: Monad m
       => Term
       -> Environment m
       -> m (Value m)
```

The identity monad

```
newtype Identity a
 = Identity { runIdentity :: a }
```

```
instance Monad Identity where
  return a = Identity a
  m >>= k  = k (runIdentity m)
```

The Maybe monad

```
instance Monad Maybe where
  return             = Just
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing
  fail s             = Nothing
```

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  m >> k  =  m >>= \_ -> k
  fail   :: String -> m a
  fail s  = error s
```

The error monad

```
instance Error e => Monad (Either e) where
  return          = Right
  Left  l >>= _   = Left l
  Right r >>= k   = k r
  fail msg        = Left (strMsg msg)
```

```
class Error a where strMsg :: String -> a
```

```
instance Error String where strMsg = id
```

There is also the MonadError class for throwing and catching errors.

More categories of monads

```
class Monad m
   => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
```

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

```
instance MonadPlus [] where ...
instance MonadPlus Maybe where ...
```

Interpreter revision: enrich error messages with positions

```
termP
 = Add (Con 1)
       (At 42 (App (Con 2) (Con 3)))
```

```
> let test t = ... interP t [] ...
> test termP
<error: 42: should be function: 2>
```

How to compose monads?

```
interp ::    Term
        ->   Env
        ->   Pos
        ->   Either String Value
```

Indication of the reader monad

Indication of the error monad

We cannot compose monads, but we can transform monads; so we transform the error monad to become (also) a reader monad.

The reader monad

```
class Monad m
   => MonadReader r m | m -> r where
  ask   :: m r
  local :: (r -> r) -> m a -> m a
```

```
instance Monad ((->) r) where
  return = const
  f >>= k = \r -> k (f r) r
```

```
instance MonadReader r ((->) r) where
  ask       = id
  local f m = m . f
```

Monad ***transformation***

```
newtype ReaderT r m a
 = ReaderT { runReaderT :: r -> m a }
```

```
instance Monad m
      => Monad (ReaderT r m) where
  return a = ...
  m >>= k = ...
  fail s = ...
```

```
instance Monad m
      => MonadReader r (ReaderT r m) where
  ask = ...
  local f m = ...
```

A monad for error messages with positions

```
import Control.Monad.Identity
import Control.Monad.Error
import Control.Monad.Reader
```

```
type M = ReaderT Position
                 (ErrorT String
                         Identity)
type Position = Int
```

```
throwErrorMsg :: String -> M a
throwErrorMsg s
 = do
      p <- ask
      fail (show p ++ ": " ++ s)
```

Data extension and selective code replacement

```
data Term = ... | At Position Term
```

```
interp (At p t) e
 = local (const p) (interp t e)
```

```
apply (Fun k) a = k a
apply f a
 = throwErrorMsg
      ("should be function: " ++ show f)
```

```
test :: Term -> Either String (Value M)
test t = runIdentity
       $ runErrorT
       $ runReaderT (interp t []) 0
```

Riddle: define a custom-made monad (only involving (->) and Either String) to survive without the monad-transformation library.

Further reading

- Omissions from P. Wadler's "The essence of functional programming":
  - CBN vs. CBV, CPS vs. monadic style
  - Equational reasoning for monads
  - ...

- David Espinosa: "*Semantic Lego*", PhD Thesis, Columbia University, 1995.
- J.E. Labra Gayo et al.: "LPS: A Language Prototyping System Using Modular Monadic Semantics", ENTCS 44(2), 2001.
- M.P. Jones: "*A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism*", Journal of Functional Programming, 1995, Predecessor paper appeared in FPCA 1993 proceedings.
- S. Peyton Jones: "*Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*", Presented at the 2000 Marktoberdorf Summer School.
- B. O'Sullivan, D. Stewart, J. Goerzen: "*Real World Haskell*", O'Reilly Media, 2008.

Thanks!
Questions and comments welcome.